World Crises Technical Report

By: The Super Seniors

Matthew Granado
Victor Pang
Benjamin Harris
Jonathan Chen
Rohan Shah
Elizabeth Hayden

Table of Contents

Executive Analysis

We've created an application that allows a user to access a database of world crises, as well as related organizations and people. The user interface is comprised of a HTML splash page, with links to other HTML pages, one for each article. Users can import new articles using a correctly-formatted XML file, export data from the application as an XML file, and search through the data on the website using single or multi-word queries.

The application is coded in Python and HTML using Google App Engine, Django templates, Javascript, CSS, and a CSS framework called bootstrap. The data lives in XML files which much conform to an XML schema generated by our classmates. We use minixsv to validate and Elementtree to parse the XML files.

1. Introduction

1.1 What is the Problem?

The goal of the project was to create an application that uses a web interface to access a database of world crises, along with related organizations and people. Users should be able to easily navigate to various crises, organizations, and people from the main splash page, and then navigate to related articles from the current page as well as search for pages using single or multi-word queries. Users should also be able to add to the database using XML files, as well as export from the database in the form of an XML file.

1.2 Why is it interesting and important?

World crises are events that not only affect those within their proximity, but necessitate international help in order to solve them. Crises such as Hurricane Katrina or the Haiti earthquakes required aid from nations across the globe in order to help those affected. It is important to stay knowledgeable about these kinds of events because it is our duty as members of mankind to help those that are less fortunate.

1.3 Why is it hard?

Creating an application that supports importing XML files in Google App Engine presents many new hurdles that we as programmers have never encountered before. Being able to dynamically draw from an XML file and process it in a way that can be displayed in HTML is a challenge when first attempting to solve a problem of this nature. Also, Google App Engine uses a variety of unique classes

and methods to implement applications; learning and applying these is no small task. From XML validation to CSS style sheets, the difficulty of the project comes from the fact that none of us have used any of the relevant tools or frameworks.

Further difficulties can be found in the import functionality. There's nothing to stop a user from importing data that is nearly identical to existing data, and indeed there shouldn't be since two different people could have very similar information. For example, George W. Bush and his father, George Bush share the same name but for one character (obviously). We can't use simple fixes to determine if the data is erroneously duplicated or actually distinct.

1.4What are the key components of your approach and results?

Our project uses Google App Engine to serve web pages dynamically generated with Django templates using data retrieved from instances of our designed and implemented GAE models that were originally populated by importing an XML instance that was validated with a pure-python validator named “minixsv” and parsed using a utility called “elementtree”. In addition, the website uses Javascript, CSS and Twitter's CSS project, “Bootstrap,” to apply style to each webpage, and the python unittest framework is used for testing.

1.5Are there any specific limitations?

When importing XML files to add to the crises database, the XML must match up with the schema used when creating the application. If any necessary information is missing or out of place, the file is rejected. Also, for the duration of this project, only Python and HTML code was allowed to create the application.

2. Design

2.1What were your design decisions?

On the user-side, we chose to create an easily-navigated web page. On the web page, current and past crises are listed along with organizations and people related to the crises. We wanted our application to be a one-stop information center for anyone that wanted to learn about these crises, as well as any ways they can get involved, so it was vital that we include information about many aspects of each event.

We wrote the application in Google App Engine. GAE is convenient because it provides easy implementation for creating a diverse, dynamic application, and the GAE servers will automatically scale your project based on user traffic and demand. GAE allows for multiple languages to write code in; we used Python 2.7.

Because we needed the ability to add more crises to the application as needed, our application can import an XML instance, based on an XML schema provided. These instances contain information needed to create new pages for each new article. The application takes the XML instance and creates a relational database, which we then use to build the web-pages for each crisis, organization, and person. XML provides a convenient format that our application can easily read and process into new pages, so long as the instance conforms to the rules of the schema.

Overall, the application should be able to take in information about new crises and display it in a user-friendly fashion. The application should be easily accessible, and easily navigated. If one wishes, he should also be able to export information from the application in the form of XML.

2.2 User Interface:

The user interface is an HTML page hosted on the GAE host servers. The URL to the main splash page is:

http://jontitan-cs373-wc.appspot.com/

The front page contains links to each type of article: crises, organizations, and people. Each article contains information needed for the user to become knowledgeable about the subject, such as origin, time, and location.

On the front page there are links to both import and export information to and from the application. The import link brings the user to a page with a “Browse” button, used to select an XML file, and an “Import” button that tells the application to process the file and add the contents to the application. The XML file must conform to the XML schema, of which the “crisis” schema is listed below:

```
<crisis id="bath_salts">
	<name>Bath Salts</name>
		<info>
			<history>Bath Salts are legal designer drugs sold as "bath salts" where users snort the bath salts for a high similar to amphetamines. The chemicals in bath salts have been around for almost a century but in the last decade bath salts have emerged in mainstream America as a legal way to get "high." The drug first came to the attention of authorities in 2010 after reports from U.S. poison centers.</history>
			<help>Raise awareness through word of mouth</help>
			<resources>More drug rehabilitation centers</resources>
			<type>Drug Epidemic</type>
			<time>
				<time></time>
				<day>21</day>
				<month>12</month>
				<year>2010</year>
				<misc>Bath salt compounds have been around for at least a century, but did not become a national crisis until 2010 when the first warning was issued.</misc>
			</time>
			<loc>
				<city></city>
				<region></region>
				<country>United States</country>
			</loc>
			<impact>
				<human>
					<deaths>0</deaths> <!-- Unsure -->
					<displaced>0</displaced>
					<injured>500</injured> <!-- Ballpark figure -->
					<missing>0</missing>
					<misc>According to a CDC report of bath salt abusers in Michigan, users not only experience psychological side effects -- there were physical ones too. In fact, 91 percent of users had neurological damage, while 77 percent experienced cardiovascular damage and 49 percent had psychological difficulties associated with the drug. Those difficulties could be severe: 37 percent of the people who suffered mental health problems reported attempting suicide or having suicidal thoughts, related to bath salts.</misc>
				</human>
				<economic>
					<amount>0</amount>
					<currency></currency>
					<misc></misc>
```

```
                        </economic>
                    </impact>
                </info>
                <ref>
                    <primaryImage>
                        <site></site>
                        <title>Bath Salts</title>

        <url>http://www.radaronline.com/sites/radaronline.com/files/imagecache/350width/bath-salts-drugs.png</url>
                        <description></description>
                    </primaryImage>
                    <image>
                        <site></site>
                        <title>Bath Salts</title>
                        <url>http://media2.wptv.com//photo/2012/05/31/Designer_drugs8cdd7e33-58d0-477a-
b9be-42b7ed4228ab0000_20120531232903_320_240.JPG</url>
                        <description>Picture of various brands of bath salts</description>
                    </image>
                    <video>
                        <site>Youtube</site>
                        <title>Bath Salts blamed for Zombie attack in Florida</title>
                        <url>http://www.youtube.com/embed/C0Q4dRzECUs</url>
                        <description>News segment on Bath Salts</description>
                    </video>
                    <social>
         <site>Facebook</site>
           <title>Bath Salts</title>
           <url>https://www.facebook.com/pages/Ban-Bath-Salts/199330996776489</url>
           <description>Facebook page advocating the criminalization of bath salts</description>
        </social>
        <ext>
         <site>DEA</site>
           <title>DEA Website</title>
           <url>http://www.justice.gov/dea/pubs/pressrel/pr102111.html</url>
           <description>Contains information about the Bath Salts epidemic</description>
        </ext>
                    <ext>
         <site>Wikipedia</site>
           <title>Methylenedioxypyrovalerone</title>
           <url>http://en.wikipedia.org/wiki/Methylenedioxypyrovalerone</url>
           <description>Wikipedia page about bath salts</description>
        </ext>
                </ref>
                <misc></misc>
                <org idref="drug_enforcement_admin"/>
                <person idref="michele_leonhart"/>
    </crisis>
```

Figure 2.1: XML Instance for a Crisis

The export option displays all information currently on the application as an XML instance in a browser. The instance conforms the schema, and can be used to import into any similar application that uses the same schema.

2.3 UML Design:

All data in the application is added to a relational database upon import. All articles are members of “World Crises”, and all information for each article is either an attribute or a member for that subject. In the diagram below, items have lines starting with diamonds that lead to members of that item. The multiplicity of the relationship between crises and their related organizations and people is represented by asterisks above the connecting lines, and the collection name is included where applicable (such as crises*, organizations*, persons*).

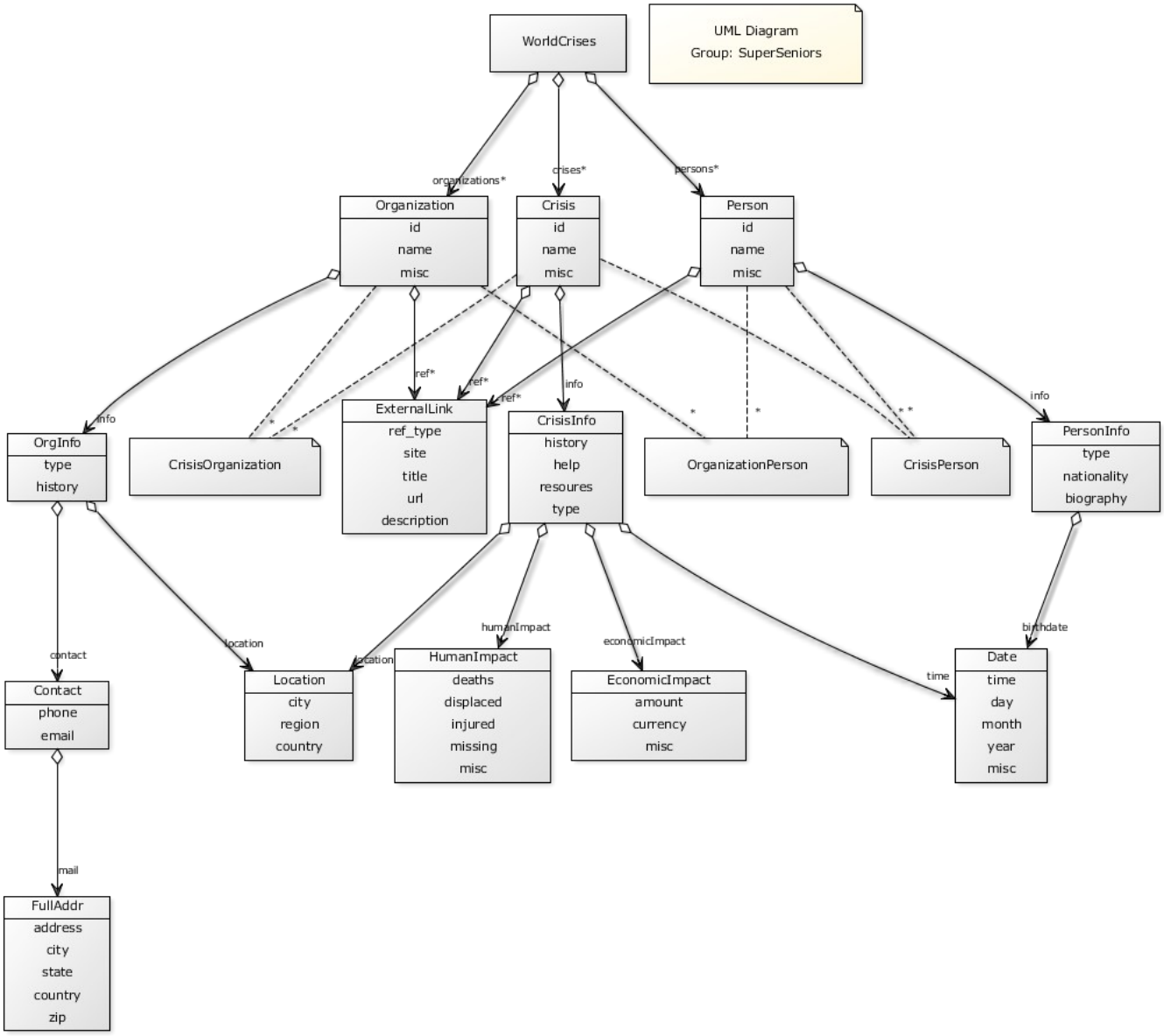


Figure 2.2: UML Diagram, showing all elements, their attributes, and their relations.

Each article is an instance of Crisis, Organization, or Person. Crisis objects have ID, name, and misc fields in addition to implicit links to a CrisisInfo instance using the collection name “info.” Crisis info objects have history, help, resources, and type attributes, and they have implicit links to a Location object using “location,” a Date object using “time,” a HumanImpact object using “humanimpact,” and an EconomicImpact object using “economicImpact.” Each of these objects also has its own set of attributes that help build a crisis article. Organization objects have the same base-level attributes as Crisis, but their OrgInfo object, linked with “info,” contains the attributes type, history, and implicit links to a Contact object (using “contact”) and a Location object (using “location”). The Contact Object has the attributes phone, email, and an implicit link (“mail”) to a FullAddr object with more attributes. Person objects have the same base-level attributes as the others, but their “info” implicit attribute links to a PersonInfo object containing biographical information and an implicit link (“birthdate”) to an instance of the Date class that contains the person's birthday information. In addition to these attributes, each of the Crisis, Organization, and Person instances contain implicit links to some number of external links, accessed with “ref.” There also exist three objects, called “CrisisOrganization,” “OrganizationPerson,” and “CrisisPerson,” that model the many-to-many relationships of each article other relevant articles.

2.4XML Design

Designing the Extensible Markup Language (XML) structure and schema was very straightforward. The project description page states very clear data requirements for each crisis, organization, or person, so our original implementation mirrored the requirements closely. Unfortunately, our implementation wasn't picked by the professor and teaching assistants, so our project was adapted to use the picked schema and XML design.

Our original XML design closely mirrored the required data, implemented a “flat” structure with low modularization, and normalized all information to strings for simplicity. The project page

requires each crisis page to contain name, kind, location, date/time, human impact, economic impact, resources needed, ways to help, and embedded external media, including images, videos, social networks, external links, and “internal” links to related organizations and people. Each organization page needed to contain name, kind, location, history, contact info, and similar embedded external media and links. Each person page needed to contain name, kind, location, and similar embedded external media and links. We chose to implement a “flat” structure in order to make our import functionality easier, and as such each crisis, organization, and person instance in an imported XML file had all of the data in one structure. Also, we specified every piece of data as a string to allow for brief explanations in the case of missing data. For example, one of our chosen crises is the conflict between Tibet and China. Understandably, it's hard to get accurate data about the economic impact of the crisis, and a simple integer value couldn't accurately reflect that. For this reason, we chose to use strings so that short strings such as “unknown” or “not applicable” could be used in addition to integer values. The flat structure made importing easier, but as a result the XML instances didn't mirror the structure of the Google App Engine models. Also, the structure could have resulted in duplicate data if our models didn't split the data up into sub-model classes.

Professor Downing and the teaching assistants chose an XML schema used by the groups “Byte Me” and “Project XML.” Their schema uses other property types, including integer, text, and string properties, and its structure more closely mirrors the Google App Engine model design. It has more layers, and very closely mirrors our GAE models. It's hard to tell if this schema allows for easier importing or readability, and it could be argued that a “flat” XML structure could still be translated into a modularized GAE model implementation.

3. Implementation

3.1 Frameworks and Tools:

Our Solution uses Google App Engine, Django, CSS, XML, and some tools such as “minixsv” and “elementtree.” Google App Engine provides a web framework with “datastore” and “blobstore” databases, and provides libraries and API's for manipulating data and handling http requests. The data store handles storing data, intuitively. The blob store holds large binary objects, or “blobs,” and keeps track of said objects with blob info entries in the data store. The Model class, provided by GAE, allowed us to specify the data “units” (models) that we wanted to put on the data store, and the “webapp2” library provides handlers for various requests and actions on the website. We used Django for its template functionality to dynamically generate HTML pages and XML files from data on the data store. Twitter's “Bootstrap” CSS “library” provided the base from which we designed our website using CSS style sheets. Mini-xsv is a pure-python XML instance validator that we used to validate imported XML instances before applying the data to the data store. Elementtree was used to parse and generate a tree structure from the XML instance during import.

3.2 Datastore:

The Datastore provided by Google App Engine provides modeling capabilities through the Model class. Our models are populated during import and put onto the datastore using the method .put(). The model structure has been described at length earlier in this report.

3.3 Import/Export:

When importing a new XML instance, we first upload the file through our import page to the blob store. Once it's on the blobstore, we clear the datastore and blobstore on the condition that “import” was selected instead of “merge.” After that, we validate the XML against the class schema using minixsv and parse the file using elementtree. Then, we build models to reflect the data and put them all on the data store. After the fact, we iterate through and update the relationships between

Crises, Organizations, and People, and finally build a data “cache” on the blobstore to allow for searching.

When exporting the datastore to XML, we query the datastore for model data and add it to lists in order. Then, we add these lists to a dictionary to be passed to Django's template engine which generates the XML file using a base XML template.

3.4 Files and Directory:

Google App Engine doesn't like subdirectories, so all of our files live either in the cs373-wc directory or its sub-directories. “Wc.py” handles all of the application's functionality, while various *.html files provide templates for Django. There are some peripheral files for javascript and CSS plug-ins.

3.5 Search:

Our search implementation is somewhat novel. Instead of using Google App Engine's built-in, experimental model-search functionality, we replicated its functionality by building a data “cache” in a text file on the blobstore during import. Once the user inputs a search query, we pull the data cache from the blobstore and search it using python's built in Boyer-Moore string search. If a match is found, the URL for that match is found only a few lines down. As far as we can tell, this implementation drastically reduces the draw on Google App Engine. We effectively avoid querying the datastore at all when searching.

3.6 Team Members:

Ben Harris and Ting-yi Chen used pair programming to implement the Google App Engine model structure, initial import functionality, the initial website design, and search functionality. Ben Harris also wrote parts of the technical report and compiled the separate pieces. Matt Granado and Victor Pang used pair programming to implement merge import, export functionality, and many-to-many relationship models for the datastore. They also wrote parts of the technical report. Rohan Shah

collected and organized the XML instance data, wrote major sections of the technical report, and sought outside design advice before re-designing the website to fit current professional web design trends. Elizabeth Hayden helped collect information for the XML data, wrote and organized the preliminary keynote presentation and parts of the technical report, and researched various functionalities for the project.

4. Evaluation

4.1 Solution Testing:

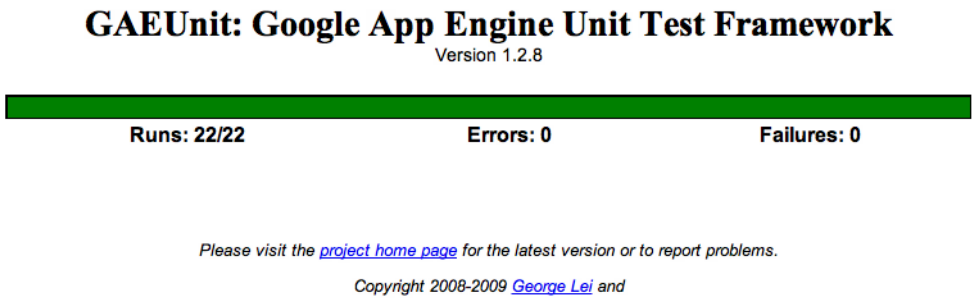
In creating the GAE web application, we needed to come up with methods to test both the back end and front end of the website. We found that just coming up with the tests was a process that required creative problem-solving, for example, how could we test that the web page that the viewer was seeing was the correct one, or that it rendered at all? Or, how do we test that the GAE models were correctly formed after ElemenTree had finished processing them? To come up with these solutions, the test writer has to have knowledge of all parts of the project in order know what needs testing, and to know how those things should be reliably tested. Using GAEUnit for unit tests, and scripts to load test the website, our team can have a quick visual way to test how new features and updates to the website affect the stability and performance of the web application.

4.2 Unit Testing:

Unit testing was accomplished using GAEUnit, which displays a web page of all the tests and whether or not they ran successfully or not. We have included a link to our GAEUnit tests on our website which runs the tests in real-time. Our first problem was to test the most obvious issue that could arise as a result of our implementation, which is: does the page even render? To test this, we created a method for each page on our website, in GAEUnit under the webTest Class. We created methods to test each webpage, e.g. Splash page, North Korea Crisis, etc... that checked to see if the web page that was rendered for that specific model actually rendered. This was accomplished by checking for a String specific to that web page, for example for our Splash page, we asserted that the string

“Splash Page” was present in the rendered HTML file. To ensure that all webTests will be created after importing our classmates XML files, we will automate the creation of these tests for each web page that will be created. This will ensure that each web page actually exists and is the correct page, removing the possibility of 404 error pages from internal links.

After writing tests for the front end, we needed to check that the GAE models were correctly made. In our test file, we created a new class, ModelTest, which included the methods to test each of the GAE models. To accomplish this, we created new model entities for crises, people, images, links, etc... Then we asserted that the attributes for each model matches what we defined it to be. The importance of this testing is that the portion of the UploadHandler which creates the GAE models is guaranteed to be correct.

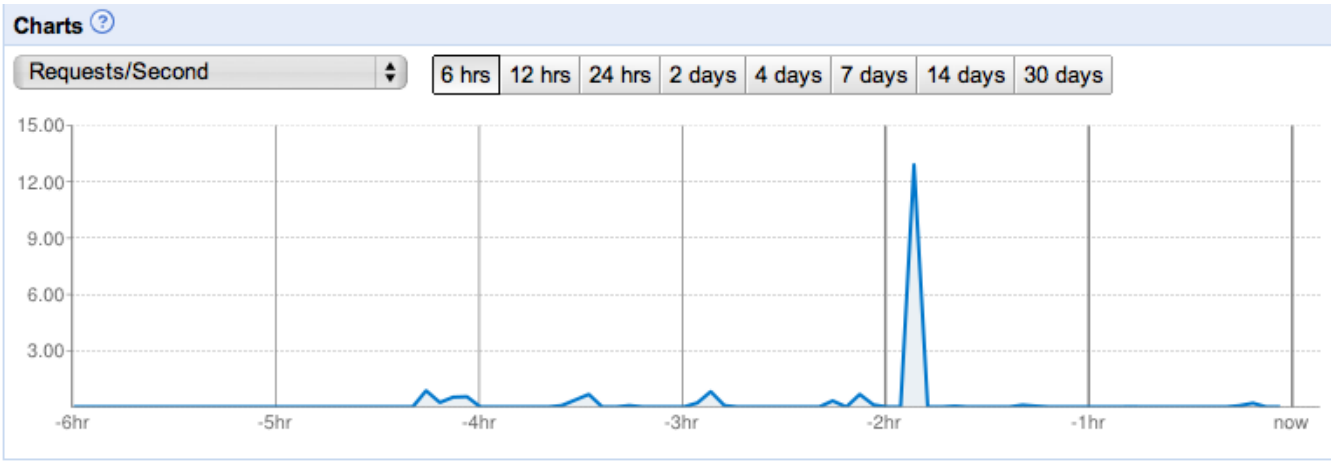


The displayed web page when the unit tests are run

Testing the part of the import process that creates the models is the only part of the import process that we discovered could be Unit Tested. We could not find a way to automate an XML upload through the ImportHandler method. We attempted writing an XML file directly to the blobstore, but that was an experimental GAE feature that was not very well documented and was too confusing to figure out. Our Import process has various asserts that makes sure that the file is an XML file, and that the file validates with our given schema. If these things are not true, then the UploadHandler redirects the user to an error page. We had to test this safety net manually, by uploading different XML files, clicking through the website to make sure all the links work, and uploading corrupt XML files to ensure we get redirected to our error page. The increased complexity of the import process required tests that were more difficult to write, and some that were impossible to write.

4.3 Performance Testing:

The performance of the website under heavy load will tell us about how well we engineered it. The admin console in GAE displays what resources are being used, and what percentage of the resource that you are alloted is being used. We used the load-test.py script provided by Google to load test our website. Running 10 threads for 10 minutes to constantly request our splash page, we reached a peak of 14 queries per second, and reached our limit of datastore read operations.



Admin Console showing all time peak of 13 QPS

Your application is at or near its free resource limits.
You should enable billing to avoid service interruption.

Billing Status: Free - Settings

Quotas re

Resource		Usage
Frontend Instance Hours	16%	4.55 of 28.00 Instance Hours
Backend Instance Hours	0%	0.00 of 9.00 Instance Hours
Datastore Stored Data	0%	0.00 of 1.00 GBytes
Logs Stored Data	0%	0.00 of 1.00 GBytes
Task Queue Stored Task Bytes	0%	0.00 of 0.49 GBytes
Blobstore Stored Data	0%	0.00 of 5.00 GBytes
Datastore Write Operations	8%	0.00 of 0.05 Million Ops
Datastore Read Operations	99%	0.05 of 0.05 Million Ops
Datastore Small Operations	0%	0.00 of 0.05 Million Ops
Outgoing Bandwidth	1%	0.01 of 1.00 GBytes
Recipients Emailed	0%	0 of 100
Stanzas Sent	0%	0 of 10,000
Channels Created	0%	0 of 100
Code and Static File Storage	0%	0.00 of 1 GBytes

Resource is currently experiencing a short-term quota limit. Learn more

Admin Console showing the Datastore Read Operations Resource has reached it's limit.

Evidently, this breaks the website, and it cannot be accessed for another 24 hours after reaching this quota.

The load testing doesn't give us very valuable information during this phase. When we implement more back-end heavy functions such as searching and import queue, this information will become valuable insofar as these functions are very resource-intensive, and pinpointing what resources are the bottlenecks will give us valuable information to help us better engineer the website to run faster.